**MULTY USER MOBILE BLUETOOTH TWO WAY TEXT CHAT**

**ABSTRACT:**

Bluetooth chatting is an innovative approach to the mobile world. This application shows use of Bluetooth in terms of chatting. Means persons can chat via Bluetooth. The main middle has just a list which has two values server and client. By selecting one of these two values, the corresponding instance is created. Middle is used to initialize the connection. It does following thing at here. First, it starts the application and search the Bluetooth device. It sends the signal to the server class. Second, it can run, pause and stop the application.. Server class goes active when it go signal from the middle class. It sends the hello world string with the string to the other devices. Client class works to respond the other Bluetooth device server.

Blue chat is a [Bluetooth](http://mobile.brothersoft.com/download/bluetooth.html) messaging. Blue chat allows you to see other Blue chat users around, ping anyone of them, and create either [private](http://mobile.brothersoft.com/download/private.html) [chat](http://mobile.brothersoft.com/download/chat.html) sessions or public messaging rooms via Bluetooth. This application allows two Android devices to carry out two-way text chat over Bluetooth. Start the application from the first screen; go to My Profile and set nickname, age, gender and hobbies. This is an optional step to provide additional information to other chat buddies. Next, select ‘Look for Friends' option which attempts to discover other users in the area. Detected user-profiles are listed as selectable boxes. While a profile is selected, Options menu can be used to view profile details, initiate a file transfer or to refresh list of detected friends.

of the classes and interfaces you will need to create Bluetooth connections:

[**Bluetooth Adapter**](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html)

Represents the local Bluetooth adapter (Bluetooth radio). The [Bluetooth Adapter](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html) is the entry-point for all Bluetooth interaction. Using this, you can discover other Bluetooth devices, query a list of bonded (paired) devices, instantiate a[BluetoothDevice](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) using a known MAC address, and create a[BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) to listen for communications from other devices.

[**Bluetooth Device**](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html)

Represents a remote Bluetooth device. Use this to request a connection with a remote device through a[BluetoothSocket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) or query information about the device such as its name, address, class, and bonding state.

[**Bluetooth Socket**](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html)

Represents the interface for a Bluetooth socket (similar to a TCP [Socket](http://developer.android.com/reference/java/net/Socket.html)). This is the connection point that allows an application to exchange data with another Bluetooth device via Input Stream and Output Stream.

[BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html)

Represents an open server socket that listens for incoming requests (similar to a TCP [Server Socket](http://developer.android.com/reference/java/net/ServerSocket.html)). In order to connect two Android devices, one device must open a server socket with this class. When a remote Bluetooth device makes a connection request to the this device, the [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) will return a connected [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) when the connection is accepted.

[**Bluetooth Class**](http://developer.android.com/reference/android/bluetooth/BluetoothClass.html)

Describes the general characteristics and capabilities of a Bluetooth device. This is a read-only set of properties that define the device's major and minor device classes and its services. However, this does not reliably describe all Bluetooth profiles and services supported by the device, but is useful as a hint to the device type.

[**Bluetooth Profile**](http://developer.android.com/reference/android/bluetooth/BluetoothProfile.html)

An interface that represents a Bluetooth profile. A Bluetooth profile is a wireless interface specification for Bluetooth-based communication between devices. An example is the Hands-Free profile.

[**BluetoothHealthAppConfiguration**](http://developer.android.com/reference/android/bluetooth/BluetoothHealthAppConfiguration.html)

Represents an application configuration that the Bluetooth Health third-party application registers to communicate with a remote Bluetooth health device.

[**BluetoothProfile.ServiceListener**](http://developer.android.com/reference/android/bluetooth/BluetoothProfile.ServiceListener.html)

An interface that notifies [Bluetooth Profile](http://developer.android.com/reference/android/bluetooth/BluetoothProfile.html) IPC clients when they have been connected to or disconnected from the service (that is, the internal service that runs a particular profile).

## **Bluetooth Permissions**

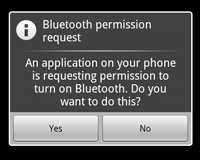
In order to use Bluetooth features in your application, you need to declare at least one of two Bluetooth permissions: [BLUETOOTH](http://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH) and [BLUETOOTH\_ADMIN](http://developer.android.com/reference/android/Manifest.permission.html" \l "BLUETOOTH_ADMIN).You must request the [BLUETOOTH](http://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH) permission in order to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data. You must request the [BLUETOOTH\_ADMIN](http://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH_ADMIN) permission in order to initiate device discovery or manipulate Bluetooth settings. Most applications need this permission solely for the ability to discover local Bluetooth devices. The other abilities granted by this permission should not be used, unless the application is a "power manager" that will modify Bluetooth settings upon user request. **Note:** If you use [BLUETOOTH\_ADMIN](http://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH_ADMIN) permission, then must also have the [BLUETOOTH](http://developer.android.com/reference/android/Manifest.permission.html#BLUETOOTH) permission.

Declare the Bluetooth permission(s) in your application manifest file. For example:

<manifest ... >  
  <uses-permission android:name="android.permission.BLUETOOTH" />  
   
</manifest>

See the [<uses-permission>](http://developer.android.com/guide/topics/manifest/uses-permission-element.html) reference for more information about declaring application permissions.

## **Setting Up Bluetooth**



**Figure 1:** The enabling Bluetooth dialog.

Before your application can communicate over Bluetooth, you need to verify that Bluetooth is supported on the device, and if so, ensure that it is enabled.

If Bluetooth is not supported, then you should gracefully disable any Bluetooth features. If Bluetooth is supported, but disabled, then you can request that the user enable Bluetooth without leaving your application. This setup is accomplished in two steps, using the Bluetooth Adapter.

The [Bluetooth Adapter](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html) is required for any and all Bluetooth activity. To get the [Bluetooth Adapter](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html), call the static [getDefaultAdapter ()](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html" \l "getDefaultAdapter())method. This returns a [Bluetooth Adapter](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html) that represents the device's own Bluetooth adapter (the Bluetooth radio). There's one Bluetooth adapter for the entire system, and your application can interact with it using this object. If [getDefaultAdapter ()](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html" \l "getDefaultAdapter()) returns null, then the device does not support Bluetooth and your story ends here. For example:

 Bluetooth Adapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();  
if (mBluetoothAdapter == null) {  
    // Device does not support Bluetooth  
}

Enable Bluetooth Next, you need to ensure that Bluetooth is enabled. Call [is Enabled()](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#isEnabled()) to check whether Bluetooth is currently enable. If this method returns false, then Bluetooth is disabled. To request that Bluetooth be enabled, call [startActivityForResult()](http://developer.android.com/reference/android/app/Activity.html" \l "startActivityForResult(android.content.Intent, int)) with the [ACTION\_REQUEST\_ENABLE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#ACTION_REQUEST_ENABLE) action Intent. This will issue a request to enable Bluetooth through the system settings (without stopping your application). For example:

if (!mBluetoothAdapter.isEnabled()) {  
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION\_REQUEST\_ENABLE);  
    startActivityForResult(enableBtIntent, REQUEST\_ENABLE\_BT);  
}

A dialog will appear requesting user permission to enable Bluetooth, as shown in Figure 1. If the user responds "Yes," the system will begin to enable Bluetooth and focus will return to your application once the process completes (or fails).

The REQUEST\_ENABLE\_BT constant passed to [startActivityForResult()](http://developer.android.com/reference/android/app/Activity.html" \l "startActivityForResult(android.content.Intent, int)) is a locally defined integer (which must be greater than 0), that the system passes back to you in your [onActivityResult()](http://developer.android.com/reference/android/app/Activity.html" \l "onActivityResult(int, int, android.content.Intent))implementation as the request Code parameter.

If enabling Bluetooth succeeds, your activity receives the [RESULT\_OK](http://developer.android.com/reference/android/app/Activity.html#RESULT_OK) result code in the[onActivityResult()](http://developer.android.com/reference/android/app/Activity.html#onActivityResult(int, int, android.content.Intent)) callback. If Bluetooth was not enabled due to an error (or the user responded "No") then the result code is [RESULT\_CANCELED](http://developer.android.com/reference/android/app/Activity.html#RESULT_CANCELED).

Optionally, your application can also listen for the [ACTION\_STATE\_CHANGED](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#ACTION_STATE_CHANGED) broadcast Intent, which the system will broadcast whenever the Bluetooth state has changed. This broadcast contains the extra fields [EXTRA\_STATE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_STATE) and [EXTRA\_PREVIOUS\_STATE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_PREVIOUS_STATE), containing the new and old Bluetooth states, respectively. Possible values for these extra fields are [STATE\_TURNING\_ON](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_TURNING_ON), [STATE\_ON](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_ON), [STATE\_TURNING\_OFF](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_TURNING_OFF), and [STATE\_OFF](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#STATE_OFF). Listening for this broadcast can be useful to detect changes made to the Bluetooth state while your app is running.

## **Finding Devices**

Using the [Bluetooth Adapter](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html), you can find remote Bluetooth devices either through device discovery or by querying the list of paired (bonded) devices.Device discovery is a scanning procedure that searches the local area for Bluetooth enabled devices and then requesting some information about each one (this is sometimes referred to as "discovering," "inquiring" or "scanning"). However, a Bluetooth device within the local area will respond to a discovery request only if it is currently enabled to be discoverable. If a device is discoverable, it will respond to the discovery request by sharing some information, such as the device name, class, and its unique MAC address. Using this information, the device performing discovery can then choose to initiate a connection to the discovered device.

Once a connection is made with a remote device for the first time, a pairing request is automatically presented to the user. When a device is paired, the basic information about that device (such as the device name, class, and MAC address) is saved and can be read using the Bluetooth APIs. Using the known MAC address for a remote device, a connection can be initiated with it at any time without performing discovery (assuming the device is within range).

Remember there is a difference between being paired and being connected. To be paired means that two devices are aware of each other's existence, have a shared link-key that can be used for authentication, and are capable of establishing an encrypted connection with each other. To be connected means that the devices currently share an

RFCOMM channel and are able to transmit data with each other. The current Android Bluetooth API's require devices to be paired before an RFCOMM connection can be established. (Pairing is automatically performed when you initiate an encrypted connection with the Bluetooth APIs.).The following sections describe how to find devices that have been paired, or discover new devices using device discovery.

### **Querying paired devices**

Before performing device discovery, its worth querying the set of paired devices to see if the desired device is already known. To do so, call get Bonded. This will return a Set of Bluetooth Devices representing paired devices. For example, you can query all paired devices and then show the name of each device to the user, using an Array Adapter:

Set<Bluetooth Device> paired Devices = mBluetoothAdapter.getBondedDevices();  
// If there are paired devices  
if (pairedDevices.size() > 0) {  
    // Loop through paired devices  
    for (Bluetooth Device device : paired Devices) {  
        // Add the name and address to an array adapter to show in a List View  
        mArrayAdapter.add(device.getName() + "\n" + device.getAddress());  
    }  
}

All that's needed from the [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) object in order to initiate a connection is the MAC address. In this example, it's saved as a part of an Array Adapter that's shown to the user. The MAC address can later be extracted in order to initiate the connection. You can learn more about creating a connection in the section about [Connecting Devices](http://developer.android.com/guide/topics/connectivity/bluetooth.html#ConnectingDevices).

### **Discovering devices**

To start discovering devices, simply call [start Discovery()](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#startDiscovery()). The process is asynchronous and the method will immediately return with a Boolean indicating whether discovery has successfully started. The discovery process usually involves an inquiry scan of about 12 seconds, followed by a page scan of each found device to retrieve its Bluetooth name.

Your application must register a Broadcast Receiver for the [ACTION\_FOUND](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#ACTION_FOUND) Intent in order to receive information about each device discovered. For each device, the system will broadcast the [ACTION\_FOUND](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#ACTION_FOUND) Intent. This Intent carries the extra fields [EXTRA\_DEVICE](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#EXTRA_DEVICE) and [EXTRA\_CLASS](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#EXTRA_CLASS), containing a [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) and a[BluetoothClass](http://developer.android.com/reference/android/bluetooth/BluetoothClass.html), respectively. For example, here's how you can register to handle the broadcast when devices are discovered:

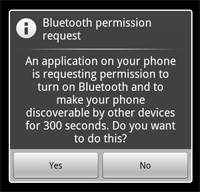
 // Create a BroadcastReceiver for ACTION\_FOUND  
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {  
    public void onReceive(Context context, Intent intent) {  
        String action = intent.getAction();  
        // When discovery finds a device  
        if (BluetoothDevice.ACTION\_FOUND.equals(action)) {  
            // Get the BluetoothDevice object from the Intent  
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA\_DEVICE);  
            // Add the name and address to an array adapter to show in a ListView  
            mArrayAdapter.add(device.getName() + "\n" + device.getAddress());  
        }  
    }  
};  
// Register the BroadcastReceiver  
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION\_FOUND);  
registerReceiver(mReceiver, filter); // Don't forget to unregister during onDestroy

All that's needed from the [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) object in order to initiate a connection is the MAC address. In this example, it's saved as a part of an Array Adapter that's shown to the user. The MAC address can later be extracted in order to initiate the connection. You can learn more about creating a connection in the section about [Connecting Devices](http://developer.android.com/guide/topics/connectivity/bluetooth.html#ConnectingDevices).

#### **Enabling discoverability**

If you would like to make the local device discoverable to other devices, call[startActivityForResult(Intent, int)](http://developer.android.com/reference/android/app/Activity.html#startActivityForResult(android.content.Intent, int)) with the [ACTION\_REQUEST\_DISCOVERABLE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#ACTION_REQUEST_DISCOVERABLE) action Intent. This will issue a request to enable discoverable mode through the system settings (without stopping your application). By default, the device will become discoverable for 120 seconds. You can define a different duration by adding the [EXTRA\_DISCOVERABLE\_DURATION](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_DISCOVERABLE_DURATION) Intent extra. The maximum duration an app can set is 3600 seconds, and a value of 0 means the device is always discoverable. Any value below 0 or above 3600 is automatically set to 120 secs). For example, this snippet sets the duration to 300:

Intent discoverableIntent = new  
Intent(BluetoothAdapter.ACTION\_REQUEST\_DISCOVERABLE);  
discoverableIntent.putExtra(BluetoothAdapter.EXTRA\_DISCOVERABLE\_DURATION, 300);  
startActivity(discoverableIntent);



**Figure 2:** The enabling discoverability dialog.

A dialog will be displayed, requesting user permission to make the device discoverable, as shown in Figure 2. If the user responds "Yes," then the device will become discoverable for the specified amount of time. Your activity will then receive a call to the [onActivityResult())](http://developer.android.com/reference/android/app/Activity.html" \l "onActivityResult(int, int, android.content.Intent)) callback, with the result code equal to the duration that the device is discoverable. If the user responded "No" or if an error occurred, the result code will beRESULT\_CANCELED. If Bluetooth has not been enabled on the device, then enabling device discoverability will automatically enable Bluetooth.

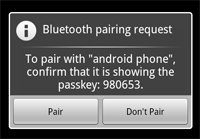
The device will silently remain in discoverable mode for the allotted time. If you would like to be notified when the discoverable mode has changed, you can register a Broadcast Receiver for the [ACTION\_SCAN\_MODE\_CHANGED](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html" \l "ACTION_SCAN_MODE_CHANGED)Intent. This will contain the extra fields [EXTRA\_SCAN\_MODE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_SCAN_MODE) and [EXTRA\_PREVIOUS\_SCAN\_MODE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#EXTRA_PREVIOUS_SCAN_MODE), which tell you the new and old scan mode, respectively. Possible values for each are[SCAN\_MODE\_CONNECTABLE\_DISCOVERABLE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#SCAN_MODE_CONNECTABLE_DISCOVERABLE), [SCAN\_MODE\_CONNECTABLE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#SCAN_MODE_CONNECTABLE), or [SCAN\_MODE\_NONE](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#SCAN_MODE_NONE), which indicate that the device is either in discoverable mode, not in discoverable mode but still able to receive connections, or not in discoverable mode and unable to receive connections, respectively.

You do not need to enable device discoverability if you will be initiating the connection to a remote device. Enabling discoverability is only necessary when you want your application to host a server socket that will accept incoming connections, because the remote devices must be able to discover the device before it can initiate the connection.

## **Connecting Devices**

In order to create a connection between your application on two devices, you must implement both the server-side and client-side mechanisms, because one device must open a server socket and the other one must initiate the connection (using the server device's MAC address to initiate a connection). The server and client are considered connected to each other when they each have a connected [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) on the same RFCOMM channel. At this point, each device can obtain input and output streams and data transfer can begin, which is discussed in the section about [Managing a Connection](http://developer.android.com/guide/topics/connectivity/bluetooth.html#ManagingAConnection). This section describes how to initiate the connection between two devices.

The server device and the client device each obtain the required [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) in different ways. The server will receive it when an incoming connection is accepted. The client will receive it when it opens an RFCOMM channel to the server.



**Figure 3:** The Bluetooth pairing dialog.

One implementation technique is to automatically prepare each device as a server, so that each one has a server socket open and listening for connections. Then either device can initiate a connection with the other and become the client. Alternatively, one device can explicitly "host" the connection and open a server socket on demand and the other device can simply initiate the connection. If the two devices have not been previously paired, then the Android framework will automatically show a pairing request notification or dialog to the user during the connection procedure, as shown in Figure 3. So when attempting to connect devices, your application does not need to be concerned about whether or not the devices are paired. Your RFCOMM connection attempt will block until the user has successfully paired, or will fail if the user rejects pairing, or if pairing fails or times out.

### **Connecting as a server**

When you want to connect two devices, one must act as a server by holding an open [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html). The purpose of the server socket is to listen for incoming connection requests and when one is accepted, provide a connected [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html). When the [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) is acquired from the [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html), the [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) can (and should) be discarded, unless you want to accept more connections. Unless you want to accept additional connections, call [close()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()).This releases the server socket and all its resources, but does not close the connected [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) that's been returned by [accept()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()). Unlike TCP/IP, RFCOMM only allows one connected client per channel at a time, so in most cases it makes sense to call [close()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()) on the [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) immediately after accepting a connected socket. The [accept()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()) call should not be executed in the main activity UI thread because it is a blocking call and will prevent any other interaction with the application. It usually makes sense to do all work with a[BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) or [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) in a new thread managed by your application. To abort a blocked call such as [accept()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()), call [close()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()) on the [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) (or [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html)) from another thread and the blocked call will immediately return. Note that all methods on a[BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) or [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) are thread-safe.

private class AcceptThread extends Thread {  
    private final BluetoothServerSocket mmServerSocket;  
   
    public AcceptThread() {  
        // Use a temporary object that is later assigned to mmServerSocket,  
        // because mmServerSocket is final  
        BluetoothServerSocket tmp = null;  
        try {  
            // MY\_UUID is the app's UUID string, also used by the client code  
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY\_UUID);  
        } catch (IOException e) { }  
        mmServerSocket = tmp;  
    }  
   
    public void run() {  
        BluetoothSocket socket = null;  
        // Keep listening until exception occurs or a socket is returned  
        while (true) {  
            try {  
                socket = mmServerSocket.accept();  
            } catch (IOException e) {  
                break;  
            }  
            // If a connection was accepted  
            if (socket != null) {  
                // Do work to manage the connection (in a separate thread)  
                manageConnectedSocket(socket);  
                mmServerSocket.close();  
                break;  
            }  
        }  
    }  
   
    /\*\* Will cancel the listening socket, and cause the thread to finish \*/  
    public void cancel() {  
        try {  
            mmServerSocket.close();  
        } catch (IOException e) { }  
    }  
}

In this example, only one incoming connection is desired, so as soon as a connection is accepted and the[Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) is acquired, the application sends the acquired [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) to a separate thread, closes the [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) and breaks the loop.

Note that when [accept()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#accept()) returns the [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html), the socket is already connected, so you should not call [connect()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) (as you do from the client-side).manageConnectedSocket() is a fictional method in the application that will initiate the thread for transferring data, which is discussed in the section about [Managing a Connection](http://developer.android.com/guide/topics/connectivity/bluetooth.html#ManagingAConnection).

You should usually close your [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) as soon as you are done listening for incoming connections. In this example, [close()](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html#close()) is called as soon as the [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) is acquired. You may also want to provide a public method in your thread that can close the private [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) in the event that you need to stop listening on the server socket.

### **Connecting as a client**

In order to initiate a connection with a remote device (a device holding an open server socket), you must first obtain a [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) object that represents the remote device. (Getting a [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) is covered in the above section about [Finding Devices](http://developer.android.com/guide/topics/connectivity/bluetooth.html#FindingDevices).) You must then use the [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html) to acquire a[BluetoothSocket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) and initiate the connection. Using the [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html), get a [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) by calling [createRfcommSocketToServiceRecord (UUID)](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html#createRfcommSocketToServiceRecord(java.util.UUID)).This initializes a [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html) that will connect to the [Bluetooth Device](http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html). The UUID passed here must match the UUID used by the server device when it opened its [BluetoothServerSocket](http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html) (with[listenUsingRfcommWithServiceRecord(String, UUID)](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#listenUsingRfcommWithServiceRecord(java.lang.String, java.util.UUID))). Using the same UUID is simply a matter of hard-coding the UUID string into your application and then referencing it from both the server and client code.

Initiate the connection by calling [connect()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()).

Upon this call, the system will perform an SDP lookup on the remote device in order to match the UUID. If the lookup is successful and the remote device accepts the connection, it will share the RFCOMM channel to use during the connection and [connect()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) will return. This method is a blocking call. If, for any reason, the connection fails or the [connect()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) method times out (after about 12 seconds), then it will throw an exception.

Because [connect()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()) is a blocking call, this connection procedure should always be performed in a thread separate from the main activity thread. Note: You should always ensure that the device is not performing device discovery when you call [connect()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#connect()). If discovery is in progress, then the connection attempt will be significantly slowed and is more likely to fail.

private class ConnectThread extends Thread {  
    private final BluetoothSocket mmSocket;  
    private final BluetoothDevice mmDevice;  
   
    public ConnectThread(BluetoothDevice device) {  
        // Use a temporary object that is later assigned to mmSocket,  
        // because mmSocket is final  
        BluetoothSocket tmp = null;  
        mmDevice = device;  
   
        // Get a BluetoothSocket to connect with the given BluetoothDevice  
        try {  
            // MY\_UUID is the app's UUID string, also used by the server code  
            tmp = device.createRfcommSocketToServiceRecord(MY\_UUID);  
        } catch (IOException e) { }  
        mmSocket = tmp;  
    }  
   
    public void run() {  
        // Cancel discovery because it will slow down the connection  
        mBluetoothAdapter.cancelDiscovery();  
   
        try {  
            // Connect the device through the socket. This will block  
            // until it succeeds or throws an exception  
            mmSocket.connect();  
        } catch (IOException connectException) {  
            // Unable to connect; close the socket and get out  
            try {  
                mmSocket.close();  
            } catch (IOException closeException) { }  
            return;  
        }  
   
        // Do work to manage the connection (in a separate thread)  
        manageConnectedSocket(mmSocket);  
    }  
   
    /\*\* Will cancel an in-progress connection, and close the socket \*/  
    public void cancel() {  
        try {  
            mmSocket.close();  
        } catch (IOException e) { }  
    }  
}

Notice that [cancel Discovery()](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#cancelDiscovery()) is called before the connection is made. You should always do this before connecting and it is safe to call without actually checking whether it is running or not (but if you do want to check, call [is Discovering()](http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#isDiscovering())).

manageConnectedSocket() is a fictional method in the application that will initiate the thread for transferring data, which is discussed in the section about [Managing a Connection](http://developer.android.com/guide/topics/connectivity/bluetooth.html#ManagingAConnection).

When you're done with your [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html), always call [close()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#close()) to clean up. Doing so will immediately close the connected socket and clean up all internal resources.

## Managing a Connection

When you have successfully connected two (or more) devices, each one will have a connected [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html). This is where the fun begins because you can share data between devices. Using the [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html), the general procedure to transfer arbitrary data is simple:

Get the [Input Stream](http://developer.android.com/reference/java/io/InputStream.html) and [Output Stream](http://developer.android.com/reference/java/io/OutputStream.html) that handle transmissions through the socket, via [get Input Stream()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#getInputStream()) and [get Output Stream()](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html#getOutputStream()), respectively.

Read and write data to the streams with [read(byte[])](http://developer.android.com/reference/java/io/InputStream.html#read(byte[])) and [write(byte[])](http://developer.android.com/reference/java/io/OutputStream.html#write(byte[])).

That's it.

There are, of course, implementation details to consider. First and foremost, you should use a dedicated thread for all stream reading and writing. This is important because both [read(byte[])](http://developer.android.com/reference/java/io/InputStream.html#read(byte[])) and [write(byte[])](http://developer.android.com/reference/java/io/OutputStream.html#write(byte[]))methods are blocking calls. [read(byte[])](http://developer.android.com/reference/java/io/InputStream.html#read(byte[])) will block until there is something to read from the stream. write does not usually block, but can block for flow control if the remote device is not calling [read(byte[])](http://developer.android.com/reference/java/io/InputStream.html#read(byte[])) quickly enough and the intermediate buffers are full. So, your main loop in the thread should be dedicated to reading from the [Input Stream](http://developer.android.com/reference/java/io/InputStream.html). A separate public method in the thread can be used to initiate writes to the [Output Stream](http://developer.android.com/reference/java/io/OutputStream.html).

Here's an example of how this might look:

private class ConnectedThread extends Thread {  
    private final BluetoothSocket mmSocket;  
    private final InputStream mmInStream;  
    private final OutputStream mmOutStream;  
   
    public ConnectedThread(BluetoothSocket socket) {  
        mmSocket = socket;  
        InputStream tmpIn = null;  
        OutputStream tmpOut = null;  
   
        // Get the input and output streams, using temp objects because  
        // member streams are final  
        try {  
            tmpIn = socket.getInputStream();  
            tmpOut = socket.getOutputStream();  
        } catch (IOException e) { }  
   
        mmInStream = tmpIn;  
        mmOutStream = tmpOut;  
    }  
   
    public void run() {  
        byte[] buffer = new byte[1024];  // buffer store for the stream  
        int bytes; // bytes returned from read()  
   
        // Keep listening to the InputStream until an exception occurs  
        while (true) {  
            try {  
                // Read from the InputStream  
                bytes = mmInStream.read(buffer);  
                // Send the obtained bytes to the UI activity  
                mHandler.obtainMessage(MESSAGE\_READ, bytes, -1, buffer)  
                        .sendToTarget();  
            } catch (IOException e) {  
                break;  
            }  
        }  
    }  
   
    /\* Call this from the main activity to send data to the remote device \*/  
    public void write(byte[] bytes) {  
        try {  
            mmOutStream.write(bytes);  
        } catch (IOException e) { }  
    }  
   
    /\* Call this from the main activity to shutdown the connection \*/  
    public void cancel() {  
        try {  
            mmSocket.close();  
        } catch (IOException e) { }  
    }  
}

The constructor acquires the necessary streams and once executed, the thread will wait for data to come through the Input Stream. When [read(byte[])](http://developer.android.com/reference/java/io/InputStream.html#read(byte[])) returns with bytes from the stream, the data is sent to the main activity using a member Handler from the parent class. Then it goes back and waits for more bytes from the stream.

Sending outgoing data is as simple as calling the thread's write() method from the main activity and passing in the bytes to be sent. This method then simply calls [write(byte[])](http://developer.android.com/reference/java/io/OutputStream.html#write(byte[])) to send the data to the remote device.

The thread's cancel() method is important so that the connection can be terminated at any time by closing the [Bluetooth Socket](http://developer.android.com/reference/android/bluetooth/BluetoothSocket.html). This should always be called when you're done using the Bluetooth connection.

**REQUIREMENT ANALYSIS**

The project involved analyzing the design of few applications so as to make the application more users friendly. To do so, it was really important to keep the navigations from one screen to the other well ordered and at the same time reducing the amount of typing the user needs to do. In order to make the application more accessible, the android version had to be chosen so that it is compatible with most of the Android devices. Hence Android 2.1 Éclair version was chosen.

**REQUIREMENT SPECIFICATION**

**Functional Requirements**

* Graphical User interface with the User.
* Provide accessibility to the application through Wi-Fi or Cellular Network.

**Software Requirements**

For developing the application the following are the Software Requirements:

1. Android Development Tools
2. Eclipse IDE 3.4 or Higher(*Resent Version*)
3. Android SDK and Eclipse Plug-ins for Android ADT (*Resent versions*).

**Operating Systems supported**

1. Windows 7
2. Windows XP
3. Windows 8

**Technologies and Languages used to Develop**

1. Android
2. Java
3. XML
4. Google Map API Services

**Debugger and Emulator**

1. Android Dalvik Debug Monitor service
2. Android Emulator(Android Virtual Device)

For running the application the following are the Software Requirements:

* Operating System: Android 2.1 or higher versions
* Network: Wi-Fi Internet or cellular Network

**Hardware Requirements**

For developing the application the following are the Hardware Requirements:

* Processor: Pentium IV or higher
* RAM: 256 MB
* Space on Hard Disk: minimum 512MB

For running the application:

* Device: Android version 2.1 and higher
* Minimum space to execute: 1.0MB